Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems

Masoud Saeida Ardekani, Pierre Sutra, Marc Shapiro

2013 IEEE 32nd International Symposium on Reliable Distributed Systems

Non-Monotonic Snapshot Isolation: scalable and strong consistency for geo-replicated transactional systems

Masoud Saeida Ardekani
Université Pierre-et-Marie Curie (UPMC-LIP6)
Paris, France
masoud.saeida-ardekani@lip6.fr

Pierre Sutra
University of Neuchâtel
Neuchâtel, Switzerland
pierre.sutra@unine.ch

Marc Shapiro
INRIA & UPMC-LIP6
Paris, France
http://lip6.fr/Marc.Shapiro/

***Abstract*—Modern cloud systems are geo-replicated to improve application latency and availability. Transactional consistency is essential for application developers; however, the corresponding concurrency control and commitment protocols are costly in a geo-replicated setting. To minimize this cost, we identify the following essential scalability properties: *(i)* only replicas updated by a transaction T make steps to execute T; *(ii)* a read-only transaction never waits for concurrent transactions and always commits; *(iii)* a transaction may read object versions committed after it started; and *(iv)* two transactions synchronize with each other only if their writes conflict. We present Non-Monotonic Snapshot Isolation (NMSI), the first strong consistency criterion to allow implementations with all four properties. We also present a practical implementation of NMSI called Jessy, which we compare experimentally against a number of well-known criteria. Our measurements show that the latency and throughput of NMSI are comparable to the weakest criterion, read-committed, and between two to fourteen times faster than well-known strong consistencies.**

I. Introduction

Cloud applications are characterized by large amounts of data that is accessed from many distributed end-points. In order to improve responsiveness and availability, cloud storage systems replicate data across several sites (data centers) located in different geographical locations. Therefore, a transaction that accesses multiple data items might have to contact several remote sites.

Many authors argue that geo-replicated systems should provide only eventual consistency [1, 2], because of the CAP impossibility result (in the presence of network faults, either consistency or availability must be forfeited [3]), and because of the high latency of strong consistency protocols in wide-area networks. However, eventual consistency is too weak for implementing some applications (e.g., banking), and is confusing for developers.

Unfortunately, classical strong consistency protocols do not scale well to high load in the wide area. Therefore, several previous works aim at designing consistency criteria that both provide meaningful guarantees to the application, and scale well [4–11]. At one end of the spectrum, strict serializability (SSER) ensures that transactions are atomic, and thus offers the strongest concurrency semantics. On the other end, read-committed (RC) guarantees only that the application accesses durable data. Under update serializability (US), read-only transactions may disagree on the order in which non-conflicting concurrent updates occur [4]. Snapshot isolation (SI) improves responsiveness of updates at the cost of the well-known write-skew anomaly [6]. The state of the art of strongly-consistent criteria for geo-replication is the weaker Parallel Snapshot Isolation (PSI), which allows transactions to take non-monotonic snapshots [9].

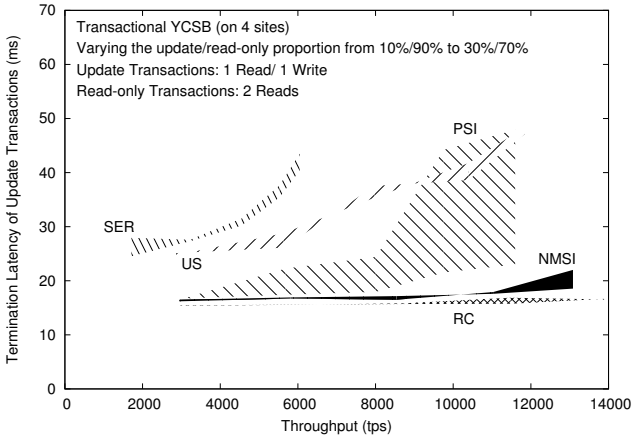


Fig. 1. Comparing the throughput and termination latency of update transactions for different protocols

Figure 1 is a preview of our experimental comparison later in this paper. For a given protocol, each point plots the throughput and latency for a given load, increasing the number of clients (left to right), and varying the proportion (from 10% to 30%, bottom to top) of update transactions.[1] Note the well-identified region of operation of each protocol, and how both latency and throughput improve with weaker criteria. Note also that in some regions PSI performance is poorer than US, and that a gap exists between PSI and RC, the weakest criterion.

Our first contribution is to identify some crucial scalability properties that explain these performance differences. For

[1] The details of the experiments are in Section V.

1060-9857/13 $26.00 © 2013 IEEE
DOI 10.1109/SRDS.2013.25

Notation	Meaning
x, y	Object
T_a, T_b	Read-only transaction
T_i, for $i \in \mathbb{N}$	Update transaction
x_i	Version of x written by T_i
$w_i(x_i)$	Transaction T_i writes x
$r_i(x_j)$	Transaction T_i reads x, written by T_j
$rs(T_i)$ / $ws(T_i)$	Read-set / write-set of transaction T_i
h	Transactional history (partially ordered)
$o_i <_h o'_j$	Operation o_i of appears before o'_j in h
$x_i \ll_h x_j$	Version order $w_i(x_i) <_h w_j(x_j)$ holds

TABLE I
NOTATIONS

instance, we show that PSI suffers because the set of versions that a transaction may read is "frozen" once the transaction starts, and it may not read more recent object versions; as a result, (i) transactions are more likely to abort because they read stale data; and (ii) even replicas of objects that are not written by a transaction must do work for that transaction.

Our second contribution is the design of a consistency criterion, named Non-Monotonic Snapshot Isolation (NMSI), that both satisfies strong safety properties, and addresses the scalability problems of PSI.

The third contribution is an implementation of NMSI, called Jessy, using dependence vectors, a novel data type that enables the efficient computation of consistent snapshots.

Our final contribution is an empirical evaluation of the scalability of NMSI, along with a careful and fair comparison against a number of classical criteria, including SER, US, SI and PSI. Figure 1 and our other experiments show that the performance of NMSI is better than the others, and is comparable to the much weaker read-committed.

The outline of this paper is as follows. We identify some bottlenecks of PSI and define our four scalability properties in Section II. We introduce NMSI in Section III. Section IV describes our protocol ensuring NMSI. Our empirical comparison is presented in Section V. We review related work in Section VI, and conclude in Section VII.

II. Scalability Properties

In this section, we first discuss informally some scalability issues of PSI. We focus on PSI because it is the state-of-the-art for geo-replicated systems, and as we saw in Figure 1, it performs better than previous strong consistency criteria. Then, we identify four crucial scalability properties.

Table I summarizes our notations; we refer to Saeida Ardekani et al. [12] for a full formal treatment. Following Bernstein et al. [13], we depict a history as a graph. For instance, in the history h_1 below, transaction T_a reads the initial versions of objects x and y, whereas T_1 and T_2 update x and y respectively.

$$h_1 = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1$$
$$\searrow\ r_a(y_0).c_a \longrightarrow r_2(y_0).w_2(y_2).c_2$$

A. Scalability Limits of PSI

In Snapshot isolation (SI), a transaction reads its own consistent snapshot, and aborts only if its writes conflict with a previously-committed concurrent transaction [6, 14]. As a consequence, read-only transactions never conflict with update transactions and always commit. Since most transactions are read-only, this improves performance considerably; indeed, SI is the default criterion of major database engines, such as Oracle or Microsoft SQL Server.

Sovran et al. [9] note that SI requires snapshots to form a monotonic sequence, necessitating global synchronization, which does not scale well. This result was refined by Saeida Ardekani et al. [15], who proved that monotonic snapshots are not compatible with genuine partial replication (defined shortly).

To address this performance issue, Sovran et al. [9] propose the alternative Parallel Snapshot Isolation (PSI), which allows the relative commit order of non-conflicting transactions to vary between replicas. This leads to an anomaly called "long forks" or *non-monotonic snapshots*. For instance, in history h_2 below, transaction T_a reads $\{x_0, y_2\}$, whereas T_b reads $\{x_1, y_0\}$. Non-monotonic snapshots were already present under US, as mentioned by Garcia-Molina and Wiederhold [4].

$$h_2 = r_a(x_0) \longrightarrow r_1(x_0).w_1(x_1).c_1 \longrightarrow r_b(x_1).c_b$$
$$r_b(y_0) \longrightarrow r_2(y_0).w_2(y_2).c_2 \longrightarrow r_a(y_2).c_a$$

Although weaker than SI, PSI's snapshots have what we call *base freshness* [16]: a transaction T_i may read only those object versions that committed *before* T_i started. As argued below, base freshness constitutes a scalability bottleneck:

1. Stale Data Reads: Consider a transaction T_i that executes in a site in North America. To access object x, which is not replicated locally, it sends a request to a replica in Europe, for a version that precedes the start of T_i. Due to the high inter-continental latency, it is likely that the version will be stale.

2. Increased Abort Rate: As a side-effect of reading stale data, the abort rate of global transactions increases. For example, consider that transaction T_i updates x concurrently to a transaction T_j, which commits while T_i is still running. The write of x by T_i is conflicting with T_j, and T_i must abort. But, this happens even if the actual read of x by T_i occurs *after* the commit time of transaction T_j.

3. Global Communication: We prove elsewhere [15, Theorem 4] that base freshness requires replicas which do not replicate data accessed by a transaction to execute steps on behalf of that transaction. And indeed, in the original PSI implementation, the transaction coordinator communicates with *all* replicas in the system [9]. Although this can be done in the background, off the critical path, this still consumes bandwidth and processing power at all replicas.

B. Scalability properties

Following the above analysis of PSI, and similar analysis of other criteria, we identify four properties as essential to scalability. In Section V, we assess empirically their relevance for several representative workloads.

1. Wait-Free Queries: A read-only transaction does not wait for concurrent transactions and always commits. This property ensures that a read-only transaction is not slowed down by synchronization, which is crucial for scalability, since most workloads exhibit a high proportion of read-only transactions.

2. Genuine Partial Replication (GPR): Replication improves both locality and availability. Full replication does not scale, as every replica must perform all updates. *Partial replication* addresses this problem, by replicating only a subset of the data at each replica. Thus, if transactions would communicate only over the minimal number of replicas, synchronization and computation overhead would be reduced. However, in the general case, the overlap of transactions cannot be predicted; therefore, many partial replication protocols perform system-wide global consensus [17, 18] or communication [9]. This negates the advantages of partial replication; hence, we require *genuine* partial replication [19], in which a transaction communicates only with the replicas that store some object accessed in the transaction. With GPR, non-conflicting transactions do not interfere with each other, and the intrinsic parallelism of a workload can be exploited.

3. Minimal Commitment Synchronization: With a strong consistency criteria, transactions are at the top of Herlihy's hierarchy [20]. On the other hand, synchronization should be avoided unless absolutely necessary, because of its direct cost, and because of the convoy effects and oscillations that it causes [21]. To keep the consensus power of transactions, while alleviating their costs, Minimal Commitment Synchronization requires that, during commitment, transaction T_i waits for transaction T_j only if T_i and T_j write-conflict.

4. Forward Freshness: Some criteria freeze the set of object versions that a transaction may read as soon as the transaction starts; a version that is committed afterwards cannot be used. A criterion supports Forward Freshness if it allows reading an object version that committed after the start of the transaction. In case of global transactions (i.e., transactions that touch several sites), this property is fundamental.

III. Non-monotonic Snapshot Isolation

NMSI addresses the problems of PSI while retaining its core properties. In the following sections, we first define NMSI, and then compare it to other consistency criteria. Like other criteria, NMSI is defined by a conjunction of safety properties.

A. Definition of NMSI

Before defining NMSI, we first introduce a dependency relation between transactions as follow:

Definition 1 (Dependency): Consider a history h and two transactions T_i and T_j. We note $T_i \rhd T_j$ when transaction T_i reads a version of x installed by T_j (i.e., $r_i(x_j)$ is in h). Transaction T_i *depends* on transaction T_j when the above relation holds by transitivity, that is, $T_i \rhd^* T_j$. Transaction T_i and T_j are *independent* if neither $T_i \rhd^* T_j$ nor $T_j \rhd^* T_i$ holds.

In order to illustrate this definition, consider history $h_3 = r_1(x_0).w_1(x_1).c_1.r_a(x_1).c_a.r_b(y_0).c_b$. In h_3, transaction T_a depends on T_1. Notice that however, even if T_1 precedes T_b in real-time, T_b does not depend on T_1 in h_3.

We now define consistent snapshots with the dependency relation. A transaction sees a consistent snapshot iff it observes the effects of all transactions it depends on [22]. Formally,

Definition 2 (Consistent snapshot): A transaction T_i in a history h observes a consistent snapshot iff, for every object x, if T_i reads version x_j, T_k writes version x_k, and T_i depends on T_k, then version x_k is followed by version x_j in the version order induced by h ($x_k \ll_h x_j$). We write $h \in$ CONS when all transactions in h observe a consistent snapshot.

To illustrate this definition, consider history $h_4 = r_1(x_0).w_1(x_1).c_1.r_2(x_1).r_2(y_0).w_2(y_2).c_2.r_a(y_2).r_a(x_0).c_a$. In this history, transaction T_a does not see a consistent snapshot: T_a depends on T_2, and T_2 also depends on T_1, but T_a does not observe the effect of T_1 (i.e., x_1).

Like PSI, NMSI prevents transactions to read non-committed data. In other words, it avoids cascading aborts:

Definition 3 (Avoiding Cascading aborts): A history h avoids cascading aborts when for every read $r_i(x_j)$ in h, operation c_j precedes $r_i(x_j)$ in h. ACA denotes the set of histories that avoid cascading aborts.

The last safety property of NMSI forbids independent write-conflicting updates to commit:

Definition 4 (Write-Conflict Freedom): A history h is write-conflict free, noted $h \in$ WCF, iff independent committed transactions never write to the same object.

The conjunction of the above properties define non-monotonic snapshot isolation:

Definition 5 (NMSI): A history h is in NMSI iff h belongs to ACA $\cap$ CONS $\cap$ WCF.

B. Comparison to Other Criteria

Table II compares NMSI to other consistency criteria, along the two axes of applicative anomalies and scalability properties.

a) Applicative Anomalies: Table II(a) compares NMSI to other criteria based on the anomalies that an application might observe. Write skew, the classical anomaly of SI, is observable under NMSI. (Cahill et al. [23] show how an application can easily avoid it). Real-time violation happens when a transaction T_i observes the effect of some transaction T_j, but does not observe the effect of all the transactions that precede T_j in real-time. This issue occurs under serializability as well; this argues that it is not considered a problem in practice. Under NMSI, an application might observe non-monotonic snapshots. This anomaly also occurs in US and PSI. Following Garcia-Molina and Wiederhold [4], we believe that this is a small price to pay for improved performance.

b) Scalability Properties: With Table II(b), we turn our attention to the scalability properties of each criterion. To make our comparison fair, we consider non-trivial implementations

(a) *Disallowed Anomalies*

	SSER	SER	US	SI	PSI	NMSI	RC
Dirty Reads	x	x	x	x	x	x	x
Non-Repeat. Reads	x	x	x	x	x	x	-
Read Skew	x	x	x	x	x	x	-
Dirty Writes	x	x	x	x	x	x	x
Lost Updates	x	x	x	x	x	x	-
Write Skew	x	x	x	-	-	-	-
Non-Monotonic Snap.	x	x	-	x	-	-	-
Real-time Violation	x	-	-	x	-	-	-

(b) *Disallowed Scalability Properties*

	SSER	SER	US	SI	PSI	NMSI	RC
GPR	x	x	-	x	x	-	-
Forw. Freshness Snap.	-	-	-	x	x	-	-
Min. Commitment Synchronization	x	x	x	-	-	-	-

TABLE II
COMPARING CONSISTENCY CRITERIA (X:*disallowed*)

of the criteria: any implementation guarantees obstruction-free updates and it accepts positively-fresh histories. Obstruction freedom for update transactions states that if a transaction does not conflict with any concurrent transaction, it eventually commits.[2] A history is positively fresh when every transaction observes *at least* the most recent snapshot of the system before it starts. Without these two progress properties, one can implement for instance SI by always reading initial versions of the objects, always committing read-only transactions, and always aborting update transaction.

Because most workloads exhibit a high proportion of read-only transactions, wait-free queries is a crucial property. Hence, we assume it in Table II(b). Saeida Ardekani et al. [15, Theorems 2 and 4] show that none of SSER, SER, SI and PSI are implementable under GPR when queries are wait-free and update transactions are obstruction free. Peluso et al. [11] show that US can combine GPR and wait-free queries. In Section IV, we show that NMSI also can conjointly satisfy these two properties. As pointed out in Section II, both PSI and SI enforce base freshness, thus disallowing forward freshness. To avoid the write-skew anomaly, SSER, SER, and US need to certify update transactions with respect to read-write and write-write conflicts. Hence, they do not provide minimal commitment synchronization.

IV. PROTOCOL

We now describe Jessy, a scalable transactional system that implements NMSI and ensures the four scalability properties defined in Section II. Because distributed locking policies do not scale [24, 25], Jessy employs deferred update replication: transactions are executed optimistically, then certified by a termination protocol. Jessy uses a novel clock mechanism to ensure that snapshots are both fresh and consistent, while preserving wait-freedom of queries and genuineness. We describe it in the next section. Due to space limitations, we defer some proofs to our companion technical report [12].

[2]We recall that in SI, PSI and NMSI, conflicting transactions are those that have write-write conflict, and in SSER, SER, and US, conflicting transactions are those that have either write-write or read-write conflicts.

A. Building Consistent Snapshots

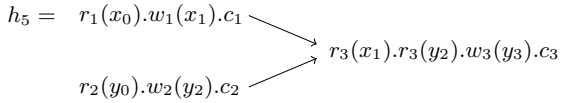
Constructing a shared snapshot object is a classical problem of distributed system literature. Nevertheless, in the context that interests us two difficulties arise: (i) multiple updates might be related to the same transaction, and (ii) the construction should be both genuine and wait-free. To achieve the above properties, Jessy makes use of a novel data type called *dependence vectors*. Each version of an object is assigned its own dependence vector. The dependence vector of some version x_i reflects all the versions read by T_i, or read by the transactions on which T_i depends, as well as the writes of T_i itself:

Definition 6 (Dependence Vector): A dependence vector is a function V that maps every read (or write) operation $o(x)$ in a history h to a vector $V(o(x)) \in \mathbb{N}^{|Objects|}$ such that:

$$V(r_i(x_0)) = 0^{|Objects|}$$
$$V(r_i(x_j)) = V(w_j(x_j))$$
$$V(w_i(x_i)) = max\ \{V(r_i(y_j)) \mid y_j \in rs(T_i)\} + \Sigma_{z_i \in ws(T_i)}\ 1_z$$

where $max\ \mathcal{V}$ is the vector containing for each dimension z, the maximal z component in the set $\mathcal{V}$, and 1_z is the vector that equals 1 on dimension z, and 0 elsewhere.

To illustrate this definition, consider history h_5 below. In this history, transactions T_1 and T_2 update objects x and y respectively, and transaction T_3 reads x then updates y. The dependence vector of $w_1(x_1)$ equals $\langle 1, 0\rangle$, and it equals $\langle 0, 1\rangle$ for $w_2(y_2)$. Since transaction T_3 reads x_1 then updates y after reading version y_2, the dependence vector of $w_3(y_3)$ equals $\langle 1, 2\rangle$.



Consider a transaction T_i and two versions x_j and y_l read by T_i. We shall say that x_j and y_l are *compatible* for T_i, written $compat(T_i, x_j, y_l)$, when both $V(r_i(x_j))[x] \geq V(r_i(y_l))[x]$ and $V(r_i(y_l))[y] \geq V(r_i(x_j))[y]$ hold. Using the compatibility relation, we can prove that dependence vectors fully characterize consistent snapshots:

Theorem 1: Consider a history h in WCF and a transaction T_i in h. Transaction T_i sees a consistent snapshot in h iff every pair of versions x_j and y_l read by T_i is compatible.

Despite that in the common case dependence vectors are sparse, they might be large for certain workloads. For instance, if transactions execute random accesses, the size of each vector tends asymptotically to the number of objects in the system. To address the above problem, Jessy employs a mechanism to approximate dependencies safely, by coarsening the granularity, grouping objects into disjoint partitions and serializing updates in a group as if it was a single larger object. We cover this mechanism in what follows.

Consider some partition $\mathcal{P}$ of $Objects$. For some object x, note $P(x)$ the partition x belongs to, and by extension, for some $S \subseteq Objects$, note $P(S)$ the set $\{\mathcal{P}(x) \mid x \in S\}$. A partition is *proper* for a history h when updates inside the same partition are serialized in h, that is, for any two writes

$w_i(x_i)$, $w_j(y_j)$ with $\mathcal{P}(x) = \mathcal{P}(y)$, either $w_i(x_i) <_h w_j(y_j)$ or the converse holds.

Now, consider some history h, and for every object x replace every operation $o_i(x)$ in h by $o_i(\mathcal{P}(x))$. We obtain a history that we note $h^{\mathcal{P}}$. The following result linked the consistency of h to the consistency of $h^{\mathcal{P}}$:

Proposition 1: Consider some history h. If $\mathcal{P}$ is a proper partition of *Objects* for h and history $h^{\mathcal{P}}$ belongs to CONS, then h is in CONS.

Given two operations $o_i(x_j)$ and $o_k(y_l)$, let us introduce relation $o_i(x_j) \leq_h^{\mathcal{P}} o_k(y_l)$ when $o_i(x_j) = o_k(y_l)$, or $o_i(x_j) <_h o_k(y_l) \wedge \mathcal{P}(x) = \mathcal{P}(y)$ holds. Based on Proposition 1, we define below a function that approximates dependencies safely:

Definition 7 (Partitioned Dependence Vector): A function PV is a partitioned dependence vector when PV maps every read (or write) operation $o(x)$ in a history h to a vector $PV(o(x)) \in \mathbb{N}^{|\mathcal{P}|}$ such that:

$$
\begin{aligned}
&PV(r_i(x_0)) = 0^{|\mathcal{P}|}\\
&PV(r_i(x_j)) = max\ \{PV(w_l(y_l)) \mid w_l(y_l) \leq_h^{\mathcal{P}} r_i(x_j)\\
&\qquad\qquad \wedge \big(\forall k : x_j \ll_h x_k \Rightarrow w_l(y_l) \leq_h^{\mathcal{P}} w_k(x_k)\big)\}\\
&PV(w_i(x_i)) = max\ \{PV(r_i(y_j)) \mid y_j \in rs(T_i)\} \cup\\
&\qquad\qquad \{PV(w_k(z_k)) : w_k(z_k) \leq_h^{\mathcal{P}} w_i(x_i)\}\\
&\qquad + \ \Sigma_{X \in \mathcal{P}(ws(T_i))}\ 1_X
\end{aligned}
$$

The first two rules of function PV are identical to the ones that would give us function V on history $h^{\mathcal{P}}$. The second part of the third rule serializes objects in the same partition

When Jessy uses partitioned dependence vectors and $\mathcal{P}$ is a proper partition for h, Theorem 1 holds for the following definition of $compat(T_i, x_j, y_l)$:

Case $\mathcal{P}(x) \neq \mathcal{P}(y)$**.** This case is identical to the definition we gave for function V. In other words, both $PV(r_i(x_j))[\mathcal{P}(x)] \geq PV(r_i(y_l))[\mathcal{P}(x)]$ and $PV(r_i(y_l))[\mathcal{P}(y)] \geq PV(r_i(x_j))[\mathcal{P}(y)]$ must hold.

Case $\mathcal{P}(x) = \mathcal{P}(y)$**.** This case deals with the fact that inside a partition writes are serialized. We have (i) if $PV(r_i(x_j))[\mathcal{P}(y)] > PV(r_i(y_l))[\mathcal{P}(y)]$ holds then $y_l = max\ \{y_k \mid w_k(y_k) \leq_h^{\mathcal{P}} w_j(x_j)\}$, or symmetrically (ii) if $PV(r_i(y_l))[\mathcal{P}(x)] > PV(r_i(x_j))[\mathcal{P}(x)]$ holds then $x_j = max\ \{x_k \mid w_k(x_k) \leq_h^{\mathcal{P}} w_l(y_l)\}$, or otherwise (iii) the predicate equals $true$.

As discussed in [16], we notice here the existence of a trade-off between the size of the vectors and the freshness of the snapshots. For instance, if x and y belong to the same partition and transaction T_i reads a version x_j, T_i cannot read a version y_l that committed after a version x_k posterior to x_j.

B. Transaction Lifetime in Jessy

Jessy is a distributed system of processes which communicate by message passing. Each process executing Jessy holds a data store that we model with variable $\mathcal{D}$. A data store contains a finite set of tuples (x, v, i), where x is an object (data item), v a value, and i a version. Jessy supports GPR, and consequently two processes may store different objects. For an object x, we shall note $replicas(x)$ the processes that store a copy of x, and by extension, $replicas(X)$ the processes that store one of the objects in X.

When a client (not modeled) executes a transaction T_i with Jessy, T_i is handled by a coordinator, denoted $coord(T_i)$. The coordinator of a transaction can be any process in the system. In what follows, $replicas(T_i)$ denotes the replica set of T_i, that is $replicas(rs(T_i) \cup ws(T_i))$.

A transaction T_i can be in one of the following four states at some process:

- *Executing*: Each non-termination operation $o_i(x)$ in T_i is executed optimistically (i.e., without synchronization with other replicas) at the transaction coordinator $coord(T_i)$. If $o_i(x)$ is a read, $coord(T_i)$ returns the corresponding value, fetched either from the local replica or a remote one. If $o_i(x)$ is a write, $coord(T_i)$ stores the corresponding update value in a local buffer, enabling *(i)* subsequent reads to observe the modification, and *(ii)* a subsequent commit to send the write-set to remote replicas.
- *Submitted*: Once all the read and write operations of T_i have executed, T_i terminates, and the coordinator submits it to the termination protocol. The protocol applies a certification test on T_i to enforce NMSI. This test ensures that if two concurrent conflicting update transactions terminate, one of them aborts.
- *Committed*/*Aborted*: When T_i enters the *Committed* state at $r \in replicas(T_i)$, its updates (if any) are applied to the local data store. If T_i aborts, T_i enters the *Aborted* state.

C. Execution Protocol

Algorithm 1 describes the execution protocol in pseudocode. Logically, it can be divided into two parts: action $remoteRead()$, executed at some process, reads an object replicated at that process in a consistent snapshot; and the coordinator $coord(T_i)$ performs actions $execute()$ to execute T_i and to buffer the updates in $up(T_i)$.

The variables of the execution protocol are: db, the local data store; $submitted$ contains locally-submitted transactions; and $committed$ (respectively $aborted$) stores committed (respectively aborted) transactions. We use the shorthand $decided$ for $committed \cup aborted$.

Upon a read request for x, $coord(T_i)$ checks against $up(T_i)$ if x has been previously updated by the same transaction; if so, it returns the corresponding value (line 13). Otherwise, $coord(T_i)$ sends a read request to the processes that replicate x (lines 16 to 17). When a process receives a read request for object x that it replicates, it returns a version of x which complies with Theorem 1 (lines 5 to 7).

Upon a write request of T_i, the process buffers the update value in $up(T_i)$ (line 10). During commitment, the updates of T_i will be sent to all replicas holding an object that is modified by T_i .

When transaction T_i terminates, it is submitted to the termination protocol (line 20). The execution protocol then waits until T_i either commits or aborts, and returns the outcome.

Algorithm 1 Execution Protocol of Jessy

```
1:  Variables:
2:      db, submitted, committed, aborted
3:
4:  remoteRead(x, T_i)
5:      pre:  received ⟨REQUEST, T_i, x⟩ from q
6:            ∃(x, v, j) ∈ db : ∀y_l ∈ rs(T_i) : compat(T_i, x_j, y_l)
7:      eff:  send ⟨REPLY, T_i, x, v⟩ to q
8:
9:  execute(WRITE, x, v, T_i)
10:     eff:  up(T_i) ← up(T_i) ∪ {(x, v, i)}
11:
12: execute(READ, x, T_i)
13:     eff:  if ∃(x, v, i) ∈ up(T_i) then return v
14:           else
15:              send ⟨REQUEST, T_i, x⟩ to replicas(x)
16:              wait until received ⟨REPLY, T_i, x, v⟩
17:              return v
18:
19: execute(TERM, T_i)
20:     eff:  submitted ← submitted ∪ {T_i}
21:           wait until T_i ∈ decided
22:           if T_i ∈ committed then return COMMIT
23:           return ABORT
24:
```

Algorithm 2 Termination Protocol of Jessy

```
1:  Variables:
2:      db, submitted, committed, aborted, Q
3:
4:  submit(T_i)
5:      pre:  T_i ∈ submitted
6:            ws(T_i) ≠ ∅
7:      eff:  AM-Cast(T_i) to replicas(ws(T_i))
8:
9:  deliver(T_i)
10:     pre:  T_i = AM-Deliver()
11:     eff:  Q ← Q ∘ ⟨T_i⟩
12:
13: vote(T_i)
14:     pre:  T_i ∈ Q \ decided
15:           ∀T_j ∈ Q, T_j <_Q T_i ⇒ T_j ∈ decided
16:     eff:  v ← certify(T_i)
17:           send ⟨VOTE, T_i, v⟩ to replicas(ws(T_i))
18:                                  ∪ {coord(T_i)}
19:
20: commit(T_i)
21:     pre:  outcome(T_i)
22:     eff:  foreach (x, v, i) in up(T_i) do
23:              if x ∈ db then db ← db ∪ {(x, v, i)}
24:           committed ← committed ∪ {T_i}
25:
26: abort(T_i)
27:     pre:  ¬outcome(T_i)
28:     eff:  aborted ← aborted ∪ {T_i}
29:
```

D. Termination Protocol

Algorithm 2 depicts the termination protocol of Jessy. It accesses the same four variables db, $submitted$ and $committed$, along with a FIFO queue named $\mathcal{Q}$.

In order to satisfy GPR, the termination protocol uses a genuine atomic multicast primitive [26, 27]. This requires that either (i) we form non-intersecting groups of replicas, and an eventual leader oracle is available in each group, or (ii) that a system-wide *reliable* failure detector is available. The latter setting allows Jessy to tolerate a disaster [28].

To terminate an update transaction T_i, $coord(T_i)$ atomic-multicasts it to every process that holds an object written by T_i. Every such process p certifies T_i by calling function $certify(T_i)$ (line 16). This function returns $true$ at process p, iff for every transaction T_j committed prior to T_i at p, if T_j write-conflicts with T_i, then T_i depends on T_j. Formally:

$$certify(T_i) \triangleq \forall T_j \in committed : ws(T_i) \cap ws(T_j) \neq \varnothing \Rightarrow T_i \rhd^* T_j$$

Under partial replication, a process p might store only a subset of the objects written by T_i, in which case p does not have enough information to decide on the outcome of T_i. Therefore, we introduce a voting phase where replicas of the objects written by T_i send the result of their certification test in a VOTE message to every process in $replicas(ws(T_i)) \cup \{coord(T_i)\}$ (lines 17 to 18).

A process can safely decide on the outcome of T_i when it has received votes from a *voting quorum* for T_i. A voting quorum Q for T_i is a set of replicas such that for every object $x \in ws(T_i)$, the set Q contains at least one of the processes replicating x. Formally, a set of processes is a voting quorum for T_i iff it belongs to $vquorum(T_i)$, defined as follows:

$$vquorum(T_i) \triangleq \{Q \subseteq \Pi \mid \forall x \in ws(T_i) : \exists j \in Q \cap replicas(x)\}$$

A process p makes use of the following (three-values) predicate $outcome(T_i)$ to determine whether some transaction T_i commits, or not:

$$\begin{array}{l} outcome(T_i) \triangleq \\ \quad \textbf{if } ws(T_i) = \varnothing \\ \quad\quad \textbf{then } true \\ \quad \textbf{else if } \forall Q \in vquorum(T_i), \exists q \in Q, \\ \quad\quad \neg received\ \langle \text{VOTE}, T, \_ \rangle\ from\ q \\ \quad\quad \textbf{then } \bot \\ \quad \textbf{else if } \exists Q \in vquorum(T_i), \forall q \in Q, \\ \quad\quad received\ \langle \text{VOTE}, T, true \rangle\ from\ q \\ \quad\quad \textbf{then } true \\ \quad \textbf{else } false \end{array}$$

To commit transaction T_i, process p first applies T_i's updates to its local data store, then p adds T_i to variable $committed$ (lines 21 to 24). If instead T_i aborts, p adds T_i to $aborted$ (lines 27 to 28).

E. Sketch of Proof

This section shows that every history accepted by Jessy is in NMSI. Then, it proves that Jessy satisfies the four scalability properties we listed in Section II-B. Both explanations are given in the broad outline, and a complete treatment is deferred to our companion technical report [12].

1) Safety Properties: Since transactions in Jessy always read committed versions of the shared objects, Jessy ensures ACA. Theorem 1 states that transactions observe consistent snapshots, hence CONS is also satisfied. It remains to show that all the histories accepted by Jessy are write-conflict free (WCF).

Assume by contradiction that two concurrent write conflicting transactions T_i and T_j both commit. Note p_i (resp. p_j) the coordinator of T_i (resp. T_j), and let x be the object on which the conflict occurs (i.e., $x \in ws(T_i) \cap ws(T_j)$). According to the definition of function $outcome$, p_i (resp. p_j) has received a yes vote from some process q_i (resp. q_j). Hence, T_i (resp. T_j) is in variable $\mathcal{Q}$ at process q_i (resp. q_j) before it sends its vote message. One can show that either once q_i sends its vote, $T_j <_{\mathcal{Q}} T_i$ holds, or once q_j sends its vote, $T_i <_{\mathcal{Q}} T_j$ holds. Assume the former case holds (the proof for the latter is symmetrical). Because of line 15 in Algorithm 2, process q_i waits until T_j is decided before sending a vote for T_i. Due to the properties of atomic multicast, and the fact that $\mathcal{Q}$ is FIFO, T_j should be committed at q_i. Thus, $certify(T_i)$ returns false at process q_i; contradiction.

2) Scalability Properties: We observe that in the case of a read-only transaction Jessy does not execute line 7 of Algorithm 2, and that the function $outcome$ always returns true. Hence, such a transaction is wait-free. As previously mentioned, a transaction is atomic multicast only to the replicas holding an object written by the transaction. Hence, the system ensures GPR. Forward freshness is reached by the $compat()$ function, and the fact that we can read the most recent committed version of an object as long as it is consistent with previous reads. Finally, a replica solely holding an object read by a transaction does not participate to the commitment. Hence, Jessy attains minimum commitment synchronization.

V. Empirical study

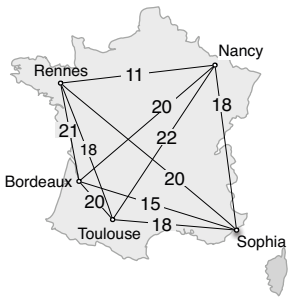
A. Implementation

We implemented Jessy as a middleware based on Algorithms 1 and 2. In our experiments, the database is an in-memory concurrent hashmap, even though Jessy normally uses BerkeleyDB. This is to minimize noise, and to focus on the scalability and synchronization costs.

We also implemented a number of replication protocols that are representative of different consistency criteria (SER, SI, US and PSI). The protocols all support partial replication; furthermore the US and SER implementations ensure GPR. The following table summarizes the criteria and the corresponding protocols:

Criterion	Protocol	Difference
SER	P-Store [19]	-
US	GMU [11]	AM-Cast instead of 2PC
SI	Serrano [17]	-
PSI	Walter [9]	AM-Cast instead of 2PC

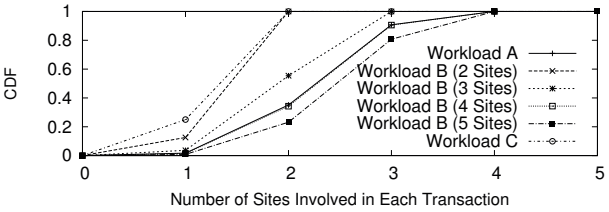
Our implementations closely follow the published specification of each protocol and are highly optimized. As they are all based on deferred update, their structure is very similar, and we were able to use the Jessy framework with relatively small variations. All our implementations use genuine atomic multicast [26, 27], even when the original used 2PC. The common structure, the use of the same multicast, and careful optimization ensure that the comparison is fair.



(a) *Sites and Latencies (in ms)*

	Key Selection Distribution	Operations	
		Read-Only Tran.	Update Tran.
A	Zipfian	4 Reads	2 Reads, 2 Updates
B	Uniform	4 Reads	3 Reads, 1 Update
C	Uniform	2 Reads	1 Read, 1 Update

(b) *Workload Types*



(c) *CDF of Number of Sites Involved in Each Transaction*

Fig. 2. Experimental Settings

The protocols all support wait-free queries, except for SER, which trades it for GPR. Since the performance of US represents an upper bound on the performance of SER with wait-free queries, this decision allows us to isolate the cost of not ensuring the property. We also implemented a (weakly-consistent) deferred-update RC, to show the maximum achievable performance. The implementation of all six protocols (SER, SI, US, PSI, NMSI and RC) takes approximately 51 kLOC in Java.

B. Setup and Benchmark

Figure 2 sums-up our experimental settings. All experiments are run on different sites of the French Grid'5000 experimental testbed [29], as illustrated in Figure 2(a). We always use four cores of machines with 2.2 GHz to 2.6 GHz processors, and a maximum heap size of 4 GB. For each server machine, two additional client machines generate the workload. Thus, there is no shared memory between clients and servers.

Every object is replicated across a multicast group of three replicas. We assume that each group as a whole is correct, i.e., it contains a majority of correct replicas. Every group contains 10^5 objects, replicated at each replica in the group, and each object has a payload size of 1 KB. Every group is replicated at a single site (no disaster tolerance). To study the scalability effects of consistency criteria in geo-replication, all our experiments are performed with global transactions. Clients are simply distributed in a uniform way between the sites.

We use the Yahoo! Cloud Serving Benchmark [30], modified to generate transactional workloads. Figure 2(b) describes the workloads used. For each workload, a client machine emulates multiple client threads in parallel, each being executed in closed loop. In all our experiments, a client machine executes at least 10^6 transactions. Figure 2(c) plots the CDF of the number of sites involved in each transaction.

The code of all the protocols, benchmarks, and scripts we used in the experiments are publicly available [31].

C. Experimental Results

We first study the impact of freshness and commitment synchronization on the latency of update transactions. Figure 3 depicts our results for workload A. The experiment is performed by varying the proportion of update/read-only transactions from 10%/90% (left) to 50%/50% (right). The load is limited so that the CPU of each replica is never saturated. The zipfian distribution is scrambled to scatter popular keys across different sites.

1. Forward Freshness: The abort ratio of update transactions, in the second graph of Figure 3, shows the effect of forward freshness. As expected, NMSI and US have the smallest abort rate, thanks to their fresher snapshots. The abort rate of US is one or two percent better than NMSI. This is mainly because NMSI is faster than US, and therefore it processes more transactions. In contrast, PSI and SI both take snapshots at the start of a transaction, resulting in an almost identical abort ratio, higher than NMSI and US. SER has the highest abort rate because in our implementation only the certification test ensures that a transaction read a consistent snapshot.

2. Minimal Commitment Synchronization: The third graph of Figure 3 studies the effect of commitment synchronization. We measure here the ratio of termination latency over solo termination latency, i.e., the time to terminate a transaction in the experiment divided by the time to terminate a transaction without contention. The ratio for RC equals 1, the optimum. This means that increasing concurrency does not increase the latency of update transactions. NMSI also has a small commitment synchronization cost. It is slightly higher for PSI, because PSI is non-genuine, and propagates when committing. This, along with its higher abort ratio, results in a termination latency increase of approximately 10 ms. SI has the highest termination latency, due to a high commitment convoy effect (because it is non-genuine), and a high abort ratio. SER low termination latency is explained by the fact that SER synchronizes both read-only and update transactions, resulting in lower thread contention than the criteria that support wait-free queries.

We now turn our attention to the impact of wait-free queries and genuine partial replication on performance. To this goal, we measure the maximal throughput of each criterion when the number of sites increases. Figure 4 depicts our results for workload B with 90% read-only transactions, and 10% update transactions.

3. Wait-Free Queries: According to Figure 4, wait-free queries have a great performance impact. Recall that our

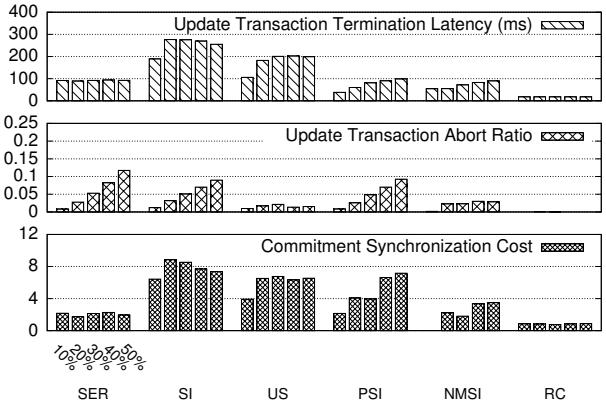


Fig. 3. Update Transaction Termination Latency (on 4 sites)

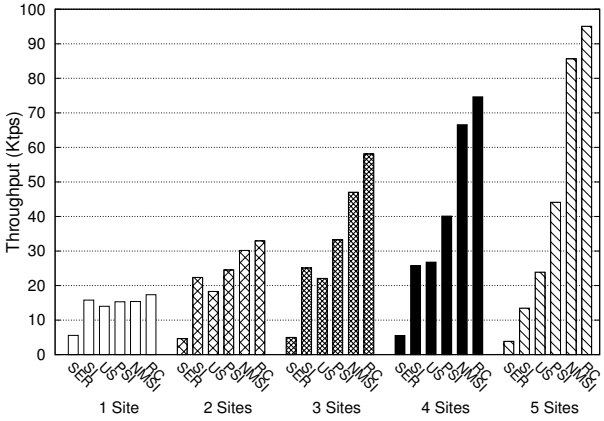


Fig. 4. Maximum Throughput of Consistency Criteria

implementation of SER favors GPR over this property. We can observe that the maximal throughput of SER, in comparison to other criteria, is at least two times lower. This assess how crucial this property is in order to scale a transactional system.

4. Genuine Partial Replication: To see the effects of GPR on system performance, we first compare PSI and NMSI. If the system consists in a single site, their throughput is almost identical. However, PSI does not scale as well as NMSI when increasing the numbers of sites: NMSI scales as linearly as RC; with five sites, its throughput is almost double of PSI. Although SI outperforms US up to three groups, it falls behind with four sites or more, and with five sites, its throughput drops substantially due to non-genuineness. Under four groups the effect is small, but with four or more sites, genuineness pays off, and US outperforms SI. Since US does not minimize commitment synchronization, its synchronization cost becomes high at 5 sites, decreasing its throughput.

We close this empirical evaluation by a detailed comparison of the scalability performance of NMSI in regard to other criteria.

5. Overall Scalability: Figure 4 shows that performance of NMSI are comparable to RC, and between two to fourteen times faster than well-known strong consistency criteria. Our

last experiment addresses the scalability of NMSI when the number of sites is constant. To this goal, we use workload C and four sites. Figure 1 plotted in Section I shows our results. The load increases from left to right. We also vary the proportion of update/read-only transactions, between 10%/90% to 30%/ 70% (bottom to top). Since workload C has few reads, SER and US do not suffer much from non-minimal commitment synchronization. For a given criterion, termination latency varies from a low end, for 10% of update transactions, to a high end for 30% of update transactions. The throughput of NMSI is similar to RC, with excellent termination latency, thanks to the combination of GPR and forward freshness. Similarly, these same properties help US to deliver better performance than PSI with a lower termination latency, when the proportion of update reaches 30%.

VI. Related Work

Strict serializability (i.e., serialization satisfying linearizability) is the strongest consistency criterion. Due to its large synchronization overhead, it had not been used at large-scale until its recent implementation by Google in Spanner [32]. Spanner is a globally distributed data store which relies on synchronized clocks to ensure consistency. On the other hand, Jessy considers a more general case where the system is partially synchronous.

Serializability (SER) is the most well-known consistency criterion for transactional systems. P-Store [19] is a genuine partial replication algorithm (for WAN environments) that ensures SER by leveraging genuine atomic multicast. Like in Jessy, read operations are performed optimistically at some replicas and update operations are applied at commit time. However, unlike Jessy, it does not ensure wait-free queries, thus it certifies read-only transactions as well.

Sciascia and Pedone [33] propose a deferred update replication protocol that supports wait-free queries and ensures SER. This approach boosts performance of SER, closing the gap with update serializability (US). However, the transaction abort ratio is higher than with US because of a more involved certification test. Besides, the system does not satisfy GPR. This last point comes from the trade-off between wait-free queries and GPR under SER, when updates are obstruction-free and histories positively-fresh [12].

Recently, Peluso et al. [34] have proposed a GPR algorithm that supports both SER and wait-free queries. This protocol works in the failure-free case and sidesteps the impossibility result by dropping obstruction-freedom for updates in certain scenarios.

A few algorithms [17, 18] offer partial replication with SI semantics. However, as a consequence of the impossibility result mentioned in Section III, none of these algorithms is genuine since no GPR system can ensure SI.

Update serializability was introduced by Garcia-Molina and Wiederhold [4], then later extended for abort transactions by Hansdah and Patnaik [5]. US provides the same guarantees as SER for update transactions, i.e., update transactions are serialized. In addition, wait-free queries can be easily implemented under US because, like in SI, they do not interfere with updates transactions. Leveraging this last property, Peluso et al. [11] have proposed recently a fast algorithm guaranteeing US for cloud systems.

Walter is a transactional key-value store designed by Sovran et al. [9] that supports Parallel Snapshot Isolation (PSI). To ensure PSI, Walter relies on a single master replication schema per object and 2PC. After a transaction commits, it has to be propagated in the background to all replicas before it becomes visible.

COPS [10] is a geo-replicated storage system that offers a strong form of causally consistent transactions. Unlike Jessy, COPS does not allow transactions to execute multiple updates. Recently, the authors have addressed this drawback [35]. However, none of the proposed solutions is strongly consistent.

VII. Conclusion

This paper introduces Non-Monotonic Snapshot Isolation (NMSI). NMSI is the first strong consistency criterion gathering the following four properties: Genuine Partial Replication, Wait-Free Queries, Forward Freshness Snapshot, and Minimal Commitment Synchronization. The conjunction of the above properties ensures that NMSI completely leverages the intrinsic parallelism of the workload and reduces the impact of concurrent transactions on each others. We also assess empirically these benefits by comparing our NMSI implementation with the implementation of several replication protocols representative of well-known criteria. Our experiments show that NMSI is close to RC (i.e, the weakest criterion) and up to two times faster than PSI.

Acknowledgments

We thank Pierpaolo Cincilla for his contribution to an early version of Jessy. We also thank Sameh Elnikety, Nuno Preguiça, Vivien Quéma, and Marek Zawirski for insightful discussions and feedbacks.

The experiments were carried out using the Grid'5000 experimental testbed, developed under INRIA ALADDIN, with support from CNRS, RENATER, and several universities, as well as other funding bodies [29].

This work is partially supported by ANR project ConcoR-DanT (ANR-10-BLAN 0208), and the European Commission's Seventh Framework Program (FP7) under grant agreement No. 318809 (LEADS).

References

[1] W. Vogels, "Eventually consistent," *ACM Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008.

[2] D. J. Abadi, "Consistency tradeoffs in modern distributed database system design," *Computer*, vol. 45, no. 2, pp. 37–42, Feb. 2012.

[3] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002.

[4] H. Garcia-Molina and G. Wiederhold, "Read-only transactions in a distributed database," *ACM Trans. Database Syst.*, vol. 7, no. 2, pp. 209–234, Jun. 1982.

[5] R. C. Hansdah and L. M. Patnaik, "Update serializability in locking," in *Lecture Notes in Computer Science*, G. Ausiello

and P. Atzeni, Eds. Springer Berlin Heidelberg, 1986, vol. 243, pp. 171–185.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *Int. Conf. on Management of Data*, New York, NY, USA, 1995, pp. 1–10.

[7] A. Adya, "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," Ph.D., MIT, Cambridge, MA, USA, Mar. 1999.

[8] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Symp. on Principles and practice of parallel programming*, ser. PPoPP '08. New York, NY, USA: ACM, 2008, pp. 175–184.

[9] Y. Sovran, R. Power, M. K. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *Symp. on Operating Systems Principles*, ser. SOSP '11, New York, NY, USA, 2011, pp. 385–400.

[10] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: scalable causal consistency for wide-area storage with cops," in *Symp. on Operating Systems Principles*. New York, NY, USA: ACM, 2011, pp. 401–416.

[11] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication," in *Int. Conf. on Distributed Computing Systems*. Macau, China: IEEE, 2012, pp. 455–465.

[12] M. Saeida Ardekani, P. Sutra, N. Preguiça, and M. Shapiro, "Non-Monotonic Snapshot Isolation," Institut National de la Recherche en Informatique et Automatique, Tech. Rep. RR-7805, Jun. 2013.

[13] P. Bernstein, V. Radzilacos, and V. Hadzilacos, *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[14] S. Elnikety, W. Zwaenepoel, and F. Pedone, "Database Replication Using Generalized Snapshot Isolation," in *Int. Symp. on Reliable Distributed Systems*, Washington, DC, USA, Oct. 2005, pp. 73–84.

[15] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguiça, "On the scalability of snapshot isolation," in *Euro-Par*, Aachen, Germany, Aug. 2013.

[16] M. Saeida Ardekani, M. Zawirski, P. Sutra, and M. Shapiro, "The space complexity of transactional interactive reads," in *Int. Work. on Hot Topics in Cloud Data Processing*. New York, New York, USA: ACM Press, Apr. 2012, pp. 1–5.

[17] D. Serrano, M. Patino-Martinez, R. Jimenez-Peris, and B. Kemme, "Boosting Database Replication Scalability through Partial Replication and 1-Copy-Snapshot-Isolation," in *Pacific Rim Int. Symp. on Dependable Computing*, Washington, DC, USA, Dec. 2007, pp. 290–297.

[18] J. E. Armendáriz-Iñigo, A. Mauch-Goya, J. R. G. de Mendívil, and F. D. Muñoz Escoí, "SIPRe: a partial database replication protocol with SI replicas," in *Symp. on Applied computing*, ser. SAC '08, New York, USA, 2008, p. 2181.

[19] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *Int. Symp. on Reliable Distributed Systems*, ser. SRDS '10, Washington, DC, USA, 2010, pp. 214–224.

[20] M. Herlihy, "Wait-free synchronization," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991.

[21] K. Birman, G. Chockler, and R. van Renesse, "Toward a Cloud Computing research agenda," *ACM SIGACT News*, vol. 40, no. 2, pp. 68–80, Jun. 2009.

[22] A. Chan and R. Gray, "Implementing Distributed Read-Only Transactions," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 205–212, Feb. 1985.

[23] M. J. Cahill, U. Röhm, and A. D. Fekete, "Serializable isolation for snapshot databases," in *Conf. on Management of Data*. New York, New York, USA: ACM Press, Jun. 2008, p. 729.

[24] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Int. Conf. on Management of data*. New York, NY, USA: ACM Press, 1996, pp. 173–182.

[25] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 551–566, 2005.

[26] R. Guerraoui and A. Schiper, "Genuine atomic multicast in asynchronous distributed systems," *Theoretical Computer Science*, vol. 254, no. 1-2, pp. 297–316, Mar. 2001.

[27] N. Schiper, P. Sutra, and F. Pedone, "Genuine versus Non-Genuine Atomic Multicast Protocols for Wide Area Networks: An Empirical Study," in *Int. Symp. on Reliable Distributed Systems*. IEEE, Sep. 2009, pp. 166–175.

[28] N. Schiper, "On Multicast Primitives in Large Networks and Partial Replication Protocols," Ph.D. dissertation, Faculty of Informatics of the University of Lugano, October 2009.

[29] Grid'5000, "Grid'5000, a scientific instrument designed to support experiment-driven research in all areas of computer science related to parallel, large-scale or distributed computing and networking," https://www.grid5000.fr/, retrieved April 2013.

[30] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Symp. on Cloud computing*. New York, NY, USA: ACM, 2010, pp. 143—-154.

[31] M. Saeida Ardekani, P. Sutra, and P. Cincilla, "https://github.com/msaeida/jessy."

[32] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally-distributed database," in *USENIX conf. on Operating Systems Design and Implementation*, Hollywood, CA, USA, Oct. 2012, pp. 251–264.

[33] D. Sciascia and F. Pedone, "Scalable Deferred Update Replication," in *Int. Conf. on Dependable Systems and Networks*. IEEE Computer Society, 2012, pp. 1–12.

[34] S. Peluso, P. Romano, and F. Quaglia, "SCORe: a scalable one-copy serializable partial replication protocol," in *Middleware*. Springer Berlin Heidelberg, Dec. 2012, pp. 456–475.

[35] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Stronger Semantics for Low-Latency Geo-Replicated Storage," in *USENIX Symp. on Network Systems Design and Implementation*, Lombard, IL, USA, 2013.